In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import pandas as pd
import os
```

Question 1

In [22]:
```
# File Path
train_data_path = './archive/Train.csv'
test_data_path = './archive/Test.csv'
train_raw_data = pd.read_csv(train_data_path)
test_raw_data = pd.read_csv(test_data_path)
```

In [23]:
```
# Data Preprocessing
# Resize Image
def resize_with_aspect_ratio(image, target_size):
    target_width, target_height = target_size
    h, w = image.shape
    scale = min(target_width / w, target_height / h)
    new_w, new_h = int(w * scale), int(h * scale)
    resize = cv2.resize(image, (new_w, new_h))
    pad_left = (target_width - new_w) // 2
    pad_right = target_width - new_w - pad_left
    pad_top = (target_height - new_h) // 2
    pad_bottom = target_height - new_h - pad_top
    padded_image = cv2.copyMakeBorder(resize, pad_top, pad_bottom, pad_left, pad
    return padded_image

# Extract ROI of each image
def preprocessImage(img_data, target_size):
    image_array = []
    class_list = []
    for index, row in img_data.iterrows():
        image_path = os.path.join('./archive', row['Path'])
        class_id = row['ClassId']
        class_list.append(class_id)
        image = cv2.imread(image_path)
        x1,y1,x2,y2 = int(row['Roi.X1']), int(row['Roi.Y1']), int(row['Roi.X2'])
        image_roi = image[y1:y2, x1:x2]
        # Convert to grayscale
        gray_image = cv2.cvtColor(image_roi, cv2.COLOR_BGR2GRAY)
        # Resize Image
        resized = resize_with_aspect_ratio(gray_image, target_size)
        # Flatten Image
        fattened_image = resized.flatten()
        image_array.append(fattened_image.reshape(-1,1))
    return image_array, class_list
```

In [24]:
```
# Training Set
TARGET_SIZE = (64, 64)
# Training: data & classes
train_data,train_class = preprocessImage(train_raw_data, TARGET_SIZE)
# Training: number of images
train_num_img = len(train_data)
```

```
train_data = np.hstack(train_data)
# Training: number of classes
train_num_class = len(set(train_class))
train_class = np.asarray(train_class)
```

In [25]:
```
# Test Set
TARGET_SIZE = (64, 64)
# Test: data & classes
test_data,test_class = preprocessImage(test_raw_data, TARGET_SIZE)
# Test: number of images
test_num_img = len(test_data)
test_data = np.hstack(test_data)
# Test: number of classes
test_num_class = len(set(test_class))
test_class = np.asarray(test_class)
```

In [151…
```
# Check Image
def check_resolution(data, size, index):
    pic = data[:,index]
    pic = pic.reshape(size[0],size[1])
    plt.imshow(pic, cmap='gray')
    plt.title("Restored Image")
    plt.axis("off")
    plt.show()
check_resolution(test_data, TARGET_SIZE, 1)
```

Restored Image

In [26]:
```
# PCA
def pca(pixel, k=0):
    m,n = pixel.shape
    # Normalize
    mean = np.mean(pixel, axis=1, keepdims=True)
    pixel_norm = pixel - mean
    # Covariance Matrix
    Q = np.dot(pixel_norm, pixel_norm.T) / n
    eigenvalues, eigenvectors = np.linalg.eigh(Q)
```

```
    # Eigenvectors & Eigenvalue
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]
    # Keep the top-k eigenvalue and eigenvector
    eigenvectors = eigenvectors[:,:k] if k != 0 else eigenvectors
    eigenvalues = eigenvalues[:k] if k != 0 else eigenvalues
    # Principle Component QV
    principal_components = np.dot(eigenvectors.T, pixel_norm)
    # Reconstruct Image
    reconstruct = np.dot(eigenvectors, principal_components) + mean

    return eigenvectors, eigenvalues, principal_components, reconstruct
```

In [27]:

```
# LDA
def lda(data, classes):
    num_classes = len(np.unique(classes))
    total_mean = np.mean(data, axis=1, keepdims=True)
    Sw = None
    Sb = None
    for c in np.unique(classes):
        indices = [i for i, val in enumerate(classes) if val == c]
        selected_data = data[:, indices]
        # Mean of Class
        class_mean = np.mean(selected_data, axis=1, keepdims=True)
        selected_data = selected_data - class_mean
        # Sw: Scatter matrix within class
        Sw_i = np.dot(selected_data, selected_data.T)
        if Sw is None:
            Sw = Sw_i
        else:
            Sw += Sw_i
        # Sb: Scatter matrix between class
        num_samples = selected_data.shape[1]
        Sb_i = num_samples * np.dot((class_mean - total_mean), (class_mean - tot
        if Sb is None:
            Sb = Sb_i
        else:
            Sb += Sb_i

    eigvals, eigvecs = np.linalg.eig(np.linalg.inv(Sw).dot(Sb))
    idx = np.argsort(eigvals)[::-1]
    eigvals = eigvals[idx]
    eigvecs = eigvecs[:, idx]

    k = num_classes - 1
    eigvals = eigvals[:k]
    eigvecs = eigvecs[:, :k]
    projection = np.dot(eigvecs.T, data)
    return eigvecs,eigvals, projection
```

PCA+LDA+KNN

In [154…

```
# Process Training data with PCA
K = 480
train_mean_pca = np.mean(train_data, axis=1, keepdims=True)
train_pca_eigen_vect,train_pca_eigen_val,train_pca_principle_components, train_r
```

In [33]:
```
# Process Training data with LDA
train_mean_lda = np.mean(train_pca_principle_components, axis=1, keepdims=True)
train_lda_eigen_vect, train_lda_eigen_val, train_lda_proj = lda(train_pca_princi
```

In [156…
```
# Process Test data with PCA
test_data = test_data - train_mean_pca
test_pca_principle_components = np.dot(train_pca_eigen_vect.T, test_data)
# Process Test data with LDA
test_lda_proj = np.dot(train_lda_eigen_vect.T, test_pca_principle_components)
```

In [157…
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score

X_train_lda_T = train_lda_proj.T
X_test_lda_T = test_lda_proj.T
X_train_lda_T = X_train_lda_T.real
X_test_lda_T = X_test_lda_T.real
# KNN Classifier
knn = KNeighborsClassifier(n_neighbors=1, weights='distance')

# Fit training data
knn.fit(X_train_lda_T, train_class)

# Predict test data
y_pred = knn.predict(X_test_lda_T)

# Accuracy
accuracy = accuracy_score(test_class, y_pred)
print("Accuracy:", accuracy)

# Confusion Matrix
conf_mat = confusion_matrix(test_class, y_pred)
print("Confusion Matrix:")
print(conf_mat)
```

```
Accuracy: 0.89944576405384
Confusion Matrix:
[[ 45   0   0 ...   0   0   0]
 [  1 678   9 ...   0   0   0]
 [  0  14 696 ...   2   0   0]
 ...
 [  0   3   1 ...  62   0   0]
 [  0   0   0 ...   0  51   0]
 [  0   0   0 ...   0   1  88]]
```

PCA+KNN

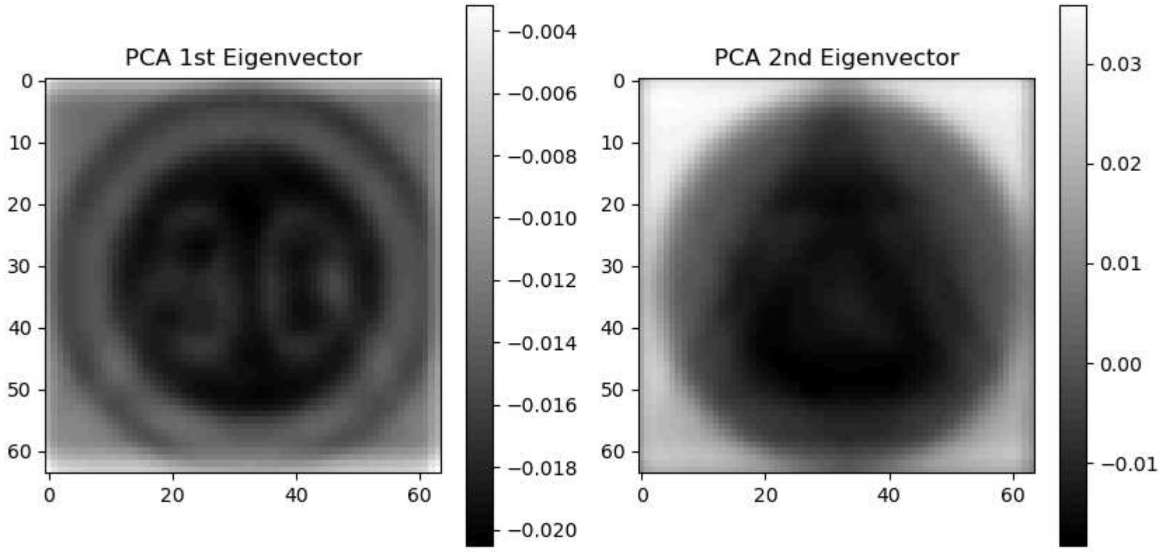
In [28]:
```
# Process Training data with PCA
K = 480
train_mean_pca = np.mean(train_data, axis=1, keepdims=True)
train_pca_eigen_vect,train_pca_eigen_val,train_pca_principle_components, train_r
```

In [29]:
```
# Process Test data with PCA
test_data = test_data - train_mean_pca
test_pca_principle_components = np.dot(train_pca_eigen_vect.T, test_data)
```

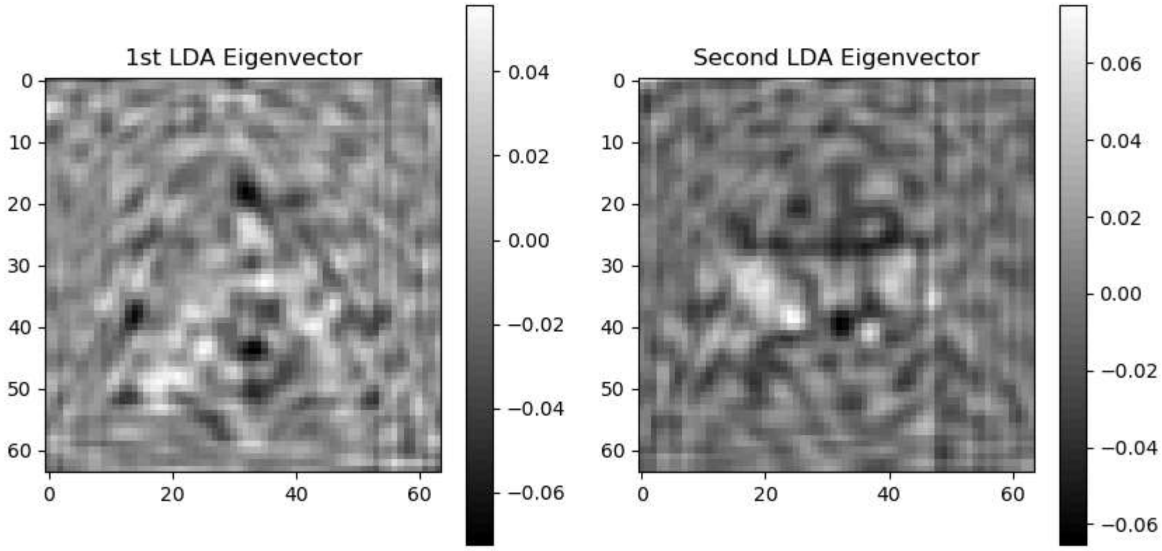
In [30]:
```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score

X_train_lda_T = train_pca_principle_components.T
X_test_lda_T = test_pca_principle_components.T
X_train_lda_T = X_train_lda_T.real
X_test_lda_T = X_test_lda_T.real
# KNN Classifier
knn = KNeighborsClassifier(n_neighbors=1, weights='distance')

# Fit training data
knn.fit(X_train_lda_T, train_class)

# Predict test data
y_pred = knn.predict(X_test_lda_T)

# Accuracy
accuracy = accuracy_score(test_class, y_pred)
print("Accuracy:", accuracy)

# Confusion Matrix
conf_mat = confusion_matrix(test_class, y_pred)
print("Confusion Matrix:")
print(conf_mat)
```

```
Accuracy: 0.47410926365795725
Confusion Matrix:
[[  4  33   6 ...   0   0   0]
 [ 14 243 168 ...   0   0   0]
 [  4  85 343 ...   7   0   1]
 ...
 [  0   2  10 ...  16   0   0]
 [  0   0   0 ...   0   4   2]
 [  0   2   2 ...   0  24  41]]
```

In [31]:
```python
Wpca= train_pca_eigen_vect
Wpca_img1 = Wpca [:, 0].reshape(64, 64)
Wpca_img2 = Wpca [:, 1].reshape(64, 64)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("PCA 1st Eigenvector")
plt.imshow(Wpca_img1, cmap='gray')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.title("PCA 2nd Eigenvector")
plt.imshow(Wpca_img2, cmap='gray')
plt.colorbar()
plt.savefig('pca_eigen.png')
plt.show()
```



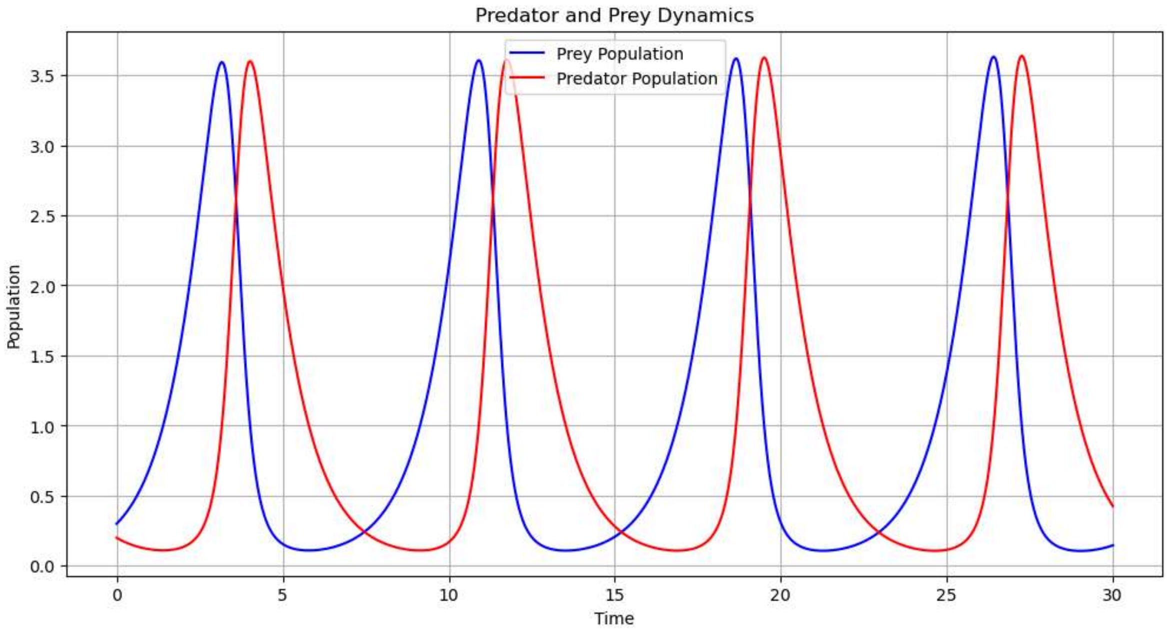
In [34]:
```python
Wlda = train_lda_eigen_vect

import matplotlib.pyplot as plt
import numpy as np

# Back-project LDA eigenvectors to original space
W_lda_back = np.dot(Wpca, Wlda).real

eigenvector_1_img = W_lda_back[:, 0].reshape(64, 64)
eigenvector_2_img = W_lda_back[:, 1].reshape(64, 64)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("1st LDA Eigenvector")
plt.imshow(eigenvector_1_img, cmap='gray')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.title("Second LDA Eigenvector")
plt.imshow(eigenvector_2_img, cmap='gray')
plt.colorbar()

plt.savefig('lda_eigen.png')
plt.show()
```



Qusetion 2

In [20]:
```
import numpy as np
import matplotlib.pyplot as plt

# Differential Equations
def preyFunc(x, y):
    return x - (x * y)

def predFunc(x, y):
    return (x * y) - y

# Initial Conditions
x0 = 0.3
y0 = 0.2

# Time range
t = np.arange(0, 30, 0.001)

# Runge-Kutta 4th Order Method
def runge_kutta(preyFunc, predFunc, x0, y0, t):
    dt = t[1] - t[0]
    x = np.zeros(len(t))
    y = np.zeros(len(t))
    x[0], y[0] = x0, y0

    for i in range(1, len(t)):
        # For prey
        k1_x = dt * preyFunc(x[i-1], y[i-1])
        k2_x = dt * preyFunc(x[i-1] + k1_x / 2, y[i-1]+ (1/2) * dt)
        k3_x = dt * preyFunc(x[i-1] + k2_x / 2, y[i-1]+ (1/2) * dt)
        k4_x = dt * preyFunc(x[i-1] + k3_x, y[i-1] + dt)
        x[i] = x[i-1] + (1/6) * k1_x + (1/3) * k2_x + (1/3) * k3_x + (1/6) * k4_

        # For predator
        k1_y = dt * predFunc(x[i-1], y[i-1])
        k2_y = dt * predFunc(x[i-1]+ (1/2) * dt, y[i-1] + k1_y / 2)
        k3_y = dt * predFunc(x[i-1]+ (1/2) * dt, y[i-1] + k2_y / 2)
        k4_y = dt * predFunc(x[i-1] + dt, y[i-1] + k3_y)
        y[i] = y[i-1] + (1/6) * k1_y + (1/3) * k2_y + (1/3) * k3_y + (1/6) * k4_

    return x, y

# Solve using RK4
x, y = runge_kutta(preyFunc, predFunc, x0, y0, t)

# Plot results
plt.figure(figsize=(12, 6))

plt.plot(t, x, label='Prey Population', color='blue')
plt.plot(t, y, label='Predator Population', color='red')
plt.title("Predator and Prey Dynamics")
plt.xlabel("Time")
plt.ylabel("Population")
plt.legend()
plt.grid()
```

```
plt.savefig("dynamic_stable.png")
plt.show()
```



In [ ]: